**Solid Principle**

It ensures that the software is modular, easy to understand, debug, and refactor

* **Single Responsibility Principle.**
* **Open-Closed Principle.**
* **Liskov Substitution Principle.**
* **Interface Segregation Principle.**
* **Dependency Inversion Principle.**

**Single Responsibility Principle:** Each class should be responsible for a single part or functionality of the system. Ex: Activity is only responsible to show the UI part and User interaction.

**Open-Closed Principle:** You should be able to extend the class behaviour, without modifying it.

Ex: If we get the new implementation in the project so better way to extend the class and work on that new class because if we change the same class then might be we can get error.

**Without Open-Closed Principle**

Public class Operation {

Public double calculate(double a1, double a2, String operationType)

{

switch(operationType) {

case “+” : {return a1+a2}

case “-” : {return a1-a2}

default:

}

}

Suppose, in future we need to implement the “\*” and “/” functionality in the same class then it might be possible to get the error in existing class.

**With Open-Closed Principle**

Public interface Operation {

Public double calculate(double a1, double a2);

}

Class AddOperation implements Operation {

@Override

Public double calculate (double a1, double a2) {

return a1+a2:

}

}

Same we will create for other operation like “-”, “\*” etc.

**Liskov Substitution Principle:** Derived classes must be substitutable fir their base classes.

**Ex:** If class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.

**Wtihout Liskov Substitution Principle**

public class Vehicle {

Public void startEngine(){}

}

Class Car extends Vehicle{

Public void startEngine(){“Start car”}

}

Class Bycle extends Vehicle{

Public void startEngine() {throw exception to start the Bycle start engine}

}

**Solution:** We should create two class VehicleWithEngine and VehicleWithOutEngine to resolve this issue.

**Interface Segregation Principle:**  Implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.

Ex: Suppose we created a interface “Salary” and its having two methods “developer” and “qa” salary. Now, if I want to use this interface for developer class then there is no use for “qa”. So in this case we should create two interface DeveloperSalaryInterface and QASalaryInterface.

**Dependency Inversion Principle:** one class should not be dependent to other classes.